VŠB – Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

Controlling Blender Software by Using Devices for Virtual Reality

Ovládání softwaru Blender pomocí zařízení pro virtuální realitu

2020 Martin Sovják

VŠB - Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

Bachelor Thesis Assignment

Student: **Martin Sovják**

Study Programme: B2647 Information and Communication Technology

Study Branch: 2612R025 Computer Science and Technology

Title: Controlling Blender Software by Using Devices for Virtual Reality
Ovládání softwaru Blender pomocí zařízení pro virtuální realitu

The thesis language: English

Description:

The goal of bachelor's thesis is creation of a tool to control an open-source software for production of 3D content from within virtual reality (VR) environment. The developed tool will be specifically used for 3D software Blender, which allows visualization of scientific data by using extension modules. Such data are generated during the solution of engineering tasks. Primary goal of the thesis will therefore be to develop the tool, which will provide the control of Blender software in VR and secondary goal will be a possibility to visualize scientific data in VR in this way.

Goals will be met by following steps:
1. Become acquainted with the issues of virtual, augmented and mixed reality including the issues of their possible use in the area of scientific data visualization.
2. Study of OpenXR standard and OpenVR interface designated for virtual reality.
3. Design a way to control Blender software and its individual parts.
4. Implementation of such design with usage of VR libraries.
5. Practical application on a specific device for VR.
6. Evaluation of achieved results.

References:

[1] Steven M. LaValle: Virtual Reality, Cambridge University Press, 2019, http://vr.cs.uiuc.edu/vrbook.pdf
[2] Egger J, Gall M, Wallner J, Boechat P, Hann A, Li X, et al. (2017) HTC Vive MeVisLab integration via OpenVR for medical applications. PLoS ONE 12(3): e0173972. doi:10.1371/journal.pone.0173972
[3] "Khronos Releases OpenXR 1.0 Specification Establishing a Foundation for the AR and VR Ecosystem". The Khronos Group. 2019
[4] https://www.khronos.org/openxr
[5] https://github.com/ValveSoftware/openvr
[6] https://github.com/cmbruns/pyopenvr
[7] https://wiki.blender.org/wiki/User:Severin/GSoC-2019/
[8] https://monado.dev/
[9] https://github.com/MARUI-PlugIn/BlenderXR/

Extent and terms of a thesis are specified in directions for its elaboration that are opened to the public on the web sites of the faculty.

Supervisor: **Ing. Lubomír Říha, Ph.D.**

Date of issue: 01.09.2019

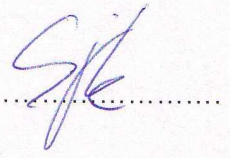
Date of submission: 30.04.2020

doc. Ing. Jan Platoš, Ph.D.
Head of Department

prof. Ing. Pavel Brandštetter, CSc.
Dean

I hereby declare that this bachelor's thesis was written by myself. I have quoted all the references I have drawn upon.

Ostrava, May 10, 2020



I would like to thank Ing. Petr Strakoš, Ph.D. for his helpful insight and advice whenever I needed it, Ing. Milan Jaroš for his guidance in the beginning and a helping hand when it was necessary and Ing. Lubomír Říha, Ph.D. for his supervision, tips and help with the formal requirements of the thesis. I would also like to thank my family and my girlfriend who greatly supported me throughout the work.

Abstrakt

V této bakalářské práci se budeme věnovat virtuální realitě v kontextu softwaru Blender. Nejdříve se zaměříme na problematiku virtuální, rozšířené a mixované reality. Poté se podíváme na standard OpenXR a rozhraní OpenVR. V druhé části práce poznatky implementujeme v podobě modulu pro ovládání prostředí 3D modelovacího softwaru Blender ve VR a osvětlíme si základy lineární algebry nezbytné pro vývoj tohoto modulu. Modul následně přizpůsobíme pro konkrétní zařízení pro virtuální realitu, kterým jsou brýle pro virtuální realitu HTC vive.

Klíčová slova: virtuální realita; Blender software; modul; plugin; ovládání; add-on; python; HTC Vive; OpenVR

Abstract

In this thesis, we will discuss virtual reality in the context of Blender software. At first, we will focus on the issues of virtual, augmented, and mixed reality. Then we will look at the OpenXR standard and OpenVR interface. In the second part of the thesis, we will implement acquired knowledge as a plugin for control of 3D modeling software Blender in VR and learn about the basics of linear algebra necessary for the development of this plugin. We will then adjust the plugin for one specific virtual reality device, which will be HTC Vive glasses for virtual reality.

Keywords: virtual reality; Blender software, plugin; control; add-on; python; HTC Vive; OpenVR

Contents

List of symbols and abbreviations **9**

List of Figures **10**

Listings **11**

1 Introduction **12**

2 Virtual, Mixed and Augmented Reality **13**
2.1 Introduction into immersive technologies 13
2.2 Usage of Virtual Reality 13
2.3 Optics in VR 15
2.4 Issues of Virtual Reality 16

3 Hardware **19**
3.1 Hardware parts 19
3.2 HTC Vive controllers 19

4 Geometry in Virtual Reality **21**
4.1 Model Geometry 21
4.2 Model Transformations 21
4.3 Coordinate spaces 26

5 OpenXR Standard **27**
5.1 Alternative software before OpenXR 27
5.2 Introduction to OpenXR 27
5.3 Goal of OpenXR 27

6 OpenVR Interface **29**

7 Blender Environment **30**
7.1 3D View 30
7.2 Shader Editor 30

8 Implementation **32**
8.1 Features 32
8.2 Application development approach 32
8.3 Handlers 34
8.4 Object Preparation 34
8.5 Blender OpenGL Wrapper 35

8.6 Obtaining VR Data . . . 37
8.7 Pynput . . . 40
8.8 Object Selection and Movement . . . 41
8.9 Vector transformations . . . 43
8.10 GPU shader module . . . 46
8.11 Blender environment adjustment . . . 48

9 Conclusion **50**

References **51**

Appendix **52**

A Add-on setup **53**

List of symbols and abbreviations

VR	–	Virtual Reality
MR	–	Mixed Reality
AR	–	Augmented Reality
UI	–	User Interface
3D	–	Three Dimensional
MRI	–	Magnetic Resonance Imaging
CT	–	Computed Tomography
ASD	–	Autism Spectrum Disorder
IPD	–	Interpupillary Distance
HMD	–	Head Mounted Display
API	–	Application Programming Interface
SDK	–	Software Development Kit
GPU	–	Graphics Processing Unit

List of Figures

1 Snell's law 16
2 Controller description 20
3 Example of a text representation of the model geometry. 22
4 Basic transformation matrices. 24
5 Two examples of local and world coordinate spaces 26
6 Blender Environment 31
7 Imported models. 35
8 Projection plane used as a menu to control the Blender environment in VR. . . . 37
9 Lines created with GPU shader. 47

Listings

1 Add-on basic example. 33
2 Example of registering a class. 34
3 Example of binding a handler onto an event. 34
4 Importing an object. 35
5 Creating a texture out of the Blender window. 36
6 Example of matrix assignment. 38
7 Pynput initialization. 40
8 Working with Pynput mouse object. 40
9 Other useful methods. The first method performs a defined number of clicks(second parameter) with a specific button(first parameter). The second method moves the mouse by x pixels on the X-axis and by y pixels on the Y-axis. 41
10 Scene ray-cast definition. 41
11 Object ray-cast definition 42
12 Calculation of mouse position from object ray-cast location result. 42
13 Parameters used in the calculation of the vectors used in `object.ray_cast` method. 44
14 Transformation of the vector for the origin parameter of the `object.ray_cast` method. 44
15 Transformation of the vector for the direction parameter of the `object.ray_cast` method. 45
16 An example of rendering a ray from raycast into 3D view. 46

1 Introduction

There is a large amount of data being computed in science and other technical fields. Such data need to be interpreted in a way that provides a natural understanding. For humans, who rely on common five senses, sight is the most important one. Therefore, visual interpretations of data are typically used. Moreover, some problems can be fully understood only if represented in 3D space. Here, virtual reality (VR) and tools for exploring the problem in VR are of special importance.

The goal of my thesis is to gain knowledge about VR and its linked fields and to develop a tool for an open-source Blender [1] software. The tool will provide a reliable way to create and observe 3D content in Blender directly from the VR environment. It will allow the visualization of scientific data if some extension modules are used. We will use the HTC Vive VR system [2] as our hardware option.

We will discuss differences between individual device types (see Section 2), their usage and issues connected to them. Then we will learn about the hardware of HTC Vive in Section 3. In Section 4 we will set foundations of linear algebra for some tasks in the implementation part. OpenXR [3] standard with OpenVR [4] application programming interface (API) are discussed in Sections 5 and 6. In the second half of the thesis we will go through the main parts of my implementation and we will demonstrate how the theory applies in practice.

2 Virtual, Mixed and Augmented Reality

2.1 Introduction into immersive technologies

Before we begin, we should ask what virtual reality is and why should we want to know more? In the last few decades, people have started wondering, how to create new experiences that would not be able to exist in the real world. With new visions for entertainment, medicine, and teaching, engineers developed first devices simulating sensory stimuli in a controlled environment and feeding them to the sensory system of humans. With different needs for certain fields, three main directions of immersive technologies formed. When a system that propagates visual stimuli to our eyes through glass or cameras, is used to project virtual objects to real surroundings around us, it is called Augmented Reality (AR). Some of the most well-known devices for AR are for example Hololens from Microsoft [5] and Glass from Google [6]. If we are put into a fully virtual space without access to the real world, we call that Virtual Reality (VR). Mixed reality (MR) uses both principles and combines a VR system with cameras and sensors of augmented reality. Some people call anything that fools our brains to alternate our perception of "real" world virtual reality. This term is not accurate, because for example in MR, the reality is not virtual, it's just altered. For more precise terminology, virtuality was introduced as a better alternative for a broader term when talking about immersive technologies. Be aware that this field is progressing fast and the examples or certain subsections might not be as relevant (or even completely forgotten) for generations to come, but general information in this thesis should still be valid.

2.2 Usage of Virtual Reality

Currently, there is a big boom in the production of VR devices and the spreading of VR experiences. With phones being able to handle VR with a decent quality and convenience while also having a low price, it is more than expected to see this area flourish. People find new ways of using VR technologies all the time and we can expect to see even more in the upcoming years. In the next few subsections, we will discuss the possible present use of VR. More details on this topic can be found in [7] and [8].

2.2.1 Healthcare

Utilization of VR for healthcare is a bit challenging because of the complexity of the data, but there is a lot of potential in the field. Currently, it is possible to process and display complex medical data acquired by MRI (Magnetic Resonance Imaging) and CT (Computed Tomography) scanners. When preparing for an operation, the VR model of an organ can be constructed from scanned data, so doctors can examine the model and plan before the procedure or it can be used for surgery trainers and simulators. We can see a new trend that is called distributed medicine, in which doctors from all around the world remotely train people and show them, how

to perform medical routines. Experiments are running on people with autism spectrum disorder (ASD), which explores how present and drawn into the VR they are.

In the future, VR could help elderly people with social connections with their families and friends in a way that feels more including, so that their mental state would be in a better condition and thus helping them with physical health as well.

2.2.2 Education

The main advantage of VR in contrast to other conventional options such as the spoken and written word is visualization in 3D, where the user is a part of a virtual 3D world. It is very helpful for fields like mathematics and sciences. If we want to learn geometry principles that are hard to interpret, this can be a good solution. VR is very helpful in cases where safety is desired when learning specific tasks. Some examples would be flight simulations, medical procedures, or nuclear power plant operations. With VR, we can achieve similar results of knowledge without the risk of injury and additional costs.

2.2.3 Science

Science is another field that can profit from access to VR. Disciplines such as physics, chemistry, biology, or engineering are very suitable for discovering new forms of research and development and can better communicate concepts that might otherwise be too abstract to understand. When the data are presented in a visual format, it is easier for the recipient to understand what is being discussed. At the same time, VR can be very interactive, so the audience can interact with the content. For these reasons VR is a great tool for scientific visualizations.

2.2.4 Gaming

The gaming industry is growing every year and it has become one of the main sources of entertainment for people across all generations. VR systems can provide a new take on game classics and use immersion to create unique experiences that conventional computers cannot produce. In the future, when sickness and fatigue problems will be solved, we can expect even more rapid growth in this area.

2.2.5 Human Interconnection

VR allows us to create synthetic worlds, in which people can connect to virtual avatars and spend time virtually together even though they are physically distant. Groups of people can meet for various reasons such as common interests, education or just to spend time with their families and friends.

VR is a great tool to make someone feel in the skin of someone else. The world still struggles with accepting social differences, races, and equality as a whole. If we put people in someone else's shoes, it helps with building empathy and understanding. For example, during wars and

economic crisis, we can simulate the environment in which the people that are affected have to live and thereby help those who are not involved in that situation to understand them. If you wanted to know what it would look like to be 15 centimeters shorter, VR systems could also answer your curiosity.

2.3 Optics in VR

To fully understand what are the challenges that VR developers have to face, we need to learn a bit about lenses and light. In VR we need to present a believable image to the user to provide a feeling as immersive as possible. If the immersion is imperfect, the user might become sick or nauseated. Lenses have to create a sharp and believable experience for the user. They need to be implemented in a way they overcome the fact that the VR system is very close to the eyes. The next sections cover more information about lenses and how the light interacts when passing through the lens surface.

2.3.1 Lenses

In some cases, VR systems do not project images on screens close to our eyes (such as CAVE), but in a lot of them, either AR (e.g. Hololens) or VR (e.g. HTC Vive), visuals are displayed on screens very close to our eyes. Our eyes are unable to focus on such a short distance and without help we would not be able to see anything clearly. Lenses solve this issue and help to focus the image right into our focal point, which is a spot where all of the parallel light rays converge. Snell's law [7] as expressed in Equation 1 explains some basic principles, how the light interacts with the lenses, and why VR systems would not work without them. To see how Snell's law is derived visit [7] and for more information about Snell's law visit [9].

2.3.2 Behavior of light

We can see light as particles, waves or rays. Each of those appear to be incompatible with the others. Thanks to modern physics it explains how those can be compatible, but we will not cover this topic further.

When light strikes any surface of a material, one of three behaviors appears. If the light enters the object and exits on the other side (it might bend or slow down), we talk about transmission. When the light enters the object and the object absorbs the energy, it is called absorption. The last option is reflection, which means that the light is deflected from the surface. If the surface perfectly reflects light, then the entering angle is the same as the exiting one, otherwise, it differs according to the refractive index. This is a very important fact for working with lenses.

Lenses work because of Snell's law [7]. We define Snell's law as

$$n_1 \, sin\theta_1 = n_2 \, sin\theta_2 \tag{1}$$

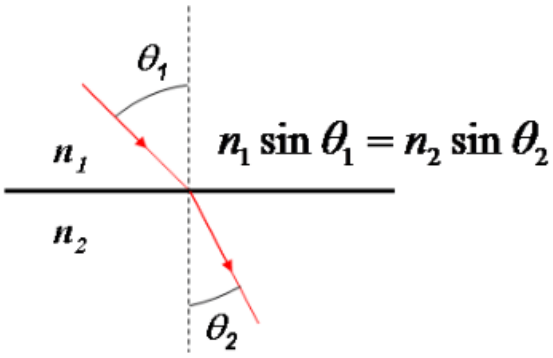


Figure 1: Snell's law, [10]

with n_1 being the refractive index of the material the light is coming from and n_2 being the index of the material the light is entering. θ_1 is the angle under which the light strikes the second material. θ_2 is the angle under which the light will continue through the second material. There are three possibilities of refraction depending on the refraction indexes of the two materials. In the case of $n_1 = n_2$, the light will not bend and it will continue under the same angle. When $n_1 > n_2$ occurs, then rays in n_2 are further from normal line than in n_1. If $n_1 < n_2$, rays in n_2 are closer to the perpendicular. Because n_1, n_2 and n_1 are given, we need to solve the Equation 1 for θ_2. We obtain this equation:

$$\theta_2 = sin^{-1}\left(\frac{n_1 sin\theta_1}{n_2}\right) \tag{2}$$

Because the range of n^{-1} is from 0 to 1, Equation 2 resolves only in the case of

$$(n_1/n_2)sin\theta_1 \leq 1 \tag{3}$$

If this condition is not met, the light will not penetrate the surface, and reflection will occur. In Figure 1, we can see a case of $n_1 < n_2$,

We can see an example of a straight surface in Figure 1, where parallel rays would exit the surface the same way. When we do not use a straight surface but we bend it to create a lens, we can accomplish parallel rays meeting in the focal point because of the properties of glass and the Snell's law. We need to account for this when building the VR system and include such lenses that can reliably provide a focused image for the user without any blur or refraction.

2.4 Issues of Virtual Reality

In this section, we will address a spectrum of issues that comes with VR, such as health problems or development problems. Visit [7] or [11] for more details.

2.4.1 Health issues

Our brain decides what is around us by a sensory system. This system consists of many parts such as eyes or vestibular organs, therefore being very complex. It processes information about our position, tilt, rotation, sounds, and more. We need to address all those parts in the VR system and design it in a way that simulates real-world stimuli as close as possible. That is not an easy task and if those very high requirements are not met, people can experience fatigue, dizziness, or even strong nausea. This happens because VR interferes with our neural structures and can cause a mismatch in the data our brain anticipates and obtains.

One of the bad examples would be vection, which is an illusion of self-motion. The eyes are reporting that the body is moving, but the balance sense is telling the brain that the body is stationary. For example, if we move in a virtual world with a controller but we do not move our body, the mismatch between the eyes and the vestibular organ might appear. In case of high movement VR experience (e.g. flying) people might lose their balance or become very disoriented and might injure themselves.

In the worst cases one might experience seizures or total blackout from flashing lights in the headset and that might happen even without any prior occurrence of epilepsy.

2.4.2 Technological issues

VR systems face many issues related to their physical construction. We will just briefly touch on this topic, for further information visit [7].

The body of every person is different. We differ in noticeable ways such as weight and height or a more subtle ways, for example distance between our eyes. The distance between eye centers called interpupillary distance (IPD) usually ranges from around 55mm to 75mm and averaging around 64mm. When designing headsets, we have to take this into account. It is important to have each lens perfectly in front of the cornea. If we do not center the lenses, it might lead to a visual mismatch in our eyes and the person using the headset might feel uncomfortable. This situation is sadly even more difficult, because here, we have not taken eye movement into account, which amplifies optical aberrations. We need to deal with optical distortion which creates the same effect as a photo taken by a fish-eyed lens would do. More issues to deal with that distort the view in the lens are coma, flare, and astigmatism. As we can see, there are many challenges when constructing a headset for VR and without proper knowledge of physics and biology it would not be possible to create a headset that would provide a proper image for the user to enjoy.

2.4.3 Subjectivity

One of the problems when developing VR systems or software is adaptation. If a person is exposed to a stimulus for longer periods or more regularly, the brain will process it differently and will get used to it. Depending on a stimuli, this may happen with any of our senses during

a wide range of time. In the dark, our eyes can adapt to it in a matter of minutes. When being exposed to a certain activity, perceptual training can occur and lead to adaptation. Adaptation is an important factor when talking about VR. Developers may develop resistance towards VR experiences that would otherwise lead to nausea for an untrained person.

Every person perceives the world uniquely. For example, if you had a color spectrum between green and yellow and asked ten people where they already see yellow and what part is still green, you would probably get ten different answers. At extremes, everyone would call it respectively green or yellow, but in the region between the extremes, the answers would vary. Scientists use experiments that measure the probability of detection [7]. It corresponds to the probability, where, in our case, a person would already say the color is yellow. This is an instance of perceptual phenomena and the study is called psychophysics. We have to take this subjectivity into account when developing any application in VR, because the application needs to produce the same result for everyone with the same input.

3 Hardware

3.1 Hardware parts

My VR system of choice is HTC Vive. It was developed by HTC and Valve and runs on SteamVR runtime. It consists of a couple of components and We will set basic terminology used in later sections of this thesis.

- Controller - A device used for getting user input.
- Headset - Or a Head-Mounted Display (HMD), contains displays and headphones for audio and video output.
- Base Station - Or a lighthouse, tracks the position of controllers and the headset.

Each part is needed for the setup. In the extreme one would be able to operate the system just with one of each device. For my thesis, we will be using two controllers. At the same time, working with only one lighthouse can be a bit challenging, because if a user restricts visibility of the controllers with his body, it will not be able to track it correctly. For better tracking, it is recommended to use two or more lighthouses placed in such a manner where every controller is visible to at least one of the lighthouses at all times. The space in which a user will move should not contain any lighthouses so they do not interfere with the movement and the possibility of injury or damage is lower.

3.2 HTC Vive controllers

When taking actions in VR, we use HTC Vive controllers. Each controller has two basic buttons, first being a trigger button located under the controller and a grip button, which can be reached from either side of the controller. Then it holds a trackpad, which is located in the middle of the controller and two menu buttons that are above and below the trackpad.

The trigger button can produce two different outputs depending on the length and pressure when pressing the button. If we press it lightly, a touch event is fired, otherwise a press event is fired. The trackpad returns the current position of the finger on it. For more information, see the Section 8.6.

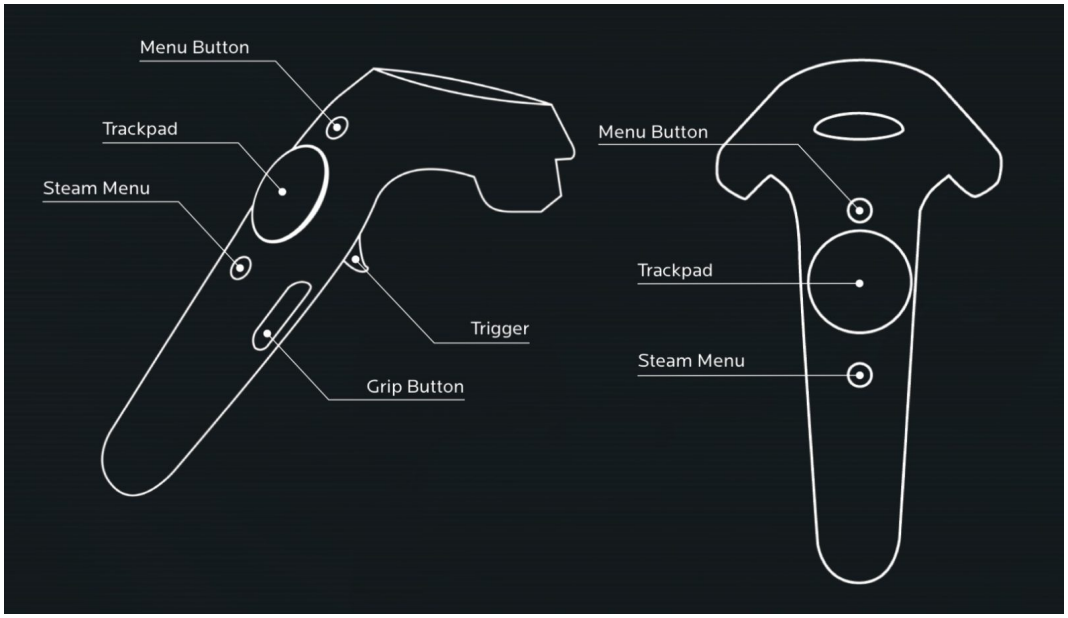


Figure 2: Controller description, [12]

4 Geometry in Virtual Reality

In this section, we will discuss the basics of VR representation of the real world, especially linear algebra needed for moving and rotating objects and data representation of objects. While working on my practical part of the thesis, I was using principals discussed in this section and I will reference it in the later sections.

4.1 Model Geometry

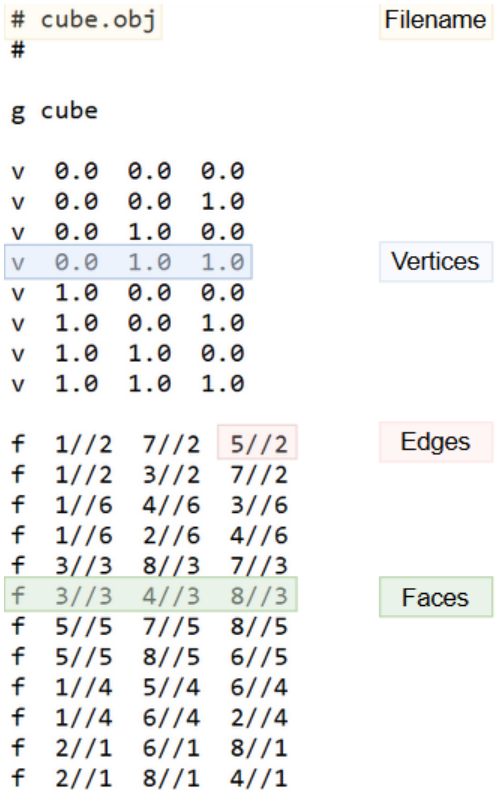
Our whole world is composed of objects. Every tree, house, or even your coffee mug is an object. We want to be able to represent objects in a way that a computer can process the information and display desired objects in a believable manner as close to reality as possible. First, we need to have a virtual world for our objects. We will use a 3D Euclidean space with Cartesian coordinates. In this world, every point has three coordinates (x, y, z). The coordinate system can differ, some use the right-hand system, others use left-handed one. In this thesis, right-handed coordinate system will be our choice. When we look at the coordinate system from the front, the X-axis points to the right, Y-axis points to the top, and Z-axis points to the front.

It is possible to approximate real models with surfaces in $\mathbb{R}^3$. If we want computers to be able to display such surfaces, they need to be represented by a finite number of vertices, between which we will define a finite number of primitives (i.e. primitive shapes) which represent an infinite number of points. The simplest and a very useful shape is a triangle. Every point inside of the triangle is specified by its three vertices represented by three $\mathbb{R}^3$ coordinates.

We need to find a way to store this information about the model. One option is to save and preserve each triangle of the model. In the case of simpler models, this method performs okay. When we want to display a more complex model (e. g. a body), it becomes very inefficient. That is why we introduce a system of vertices, edges, and faces. Vertex is a specific point in the $\mathbb{R}^3$, an edge connects two such points and three edges, which make a triangle (because we chose to use triangles as our primitive), create a face. This way, we only have to save each vertex and each edge once. That means no duplication in contrast to saving all of the vertices of each face. If we want to store such information, we write down all the vertices that our model has. Then we decide which of the vertices connect as edges and which edges form into faces. Edges are often not used so the record is as short as possible. An example of such representation is in Figure 3.

4.2 Model Transformations

If we want to move or rotate our desired model in a virtual 3D space, we need to transform every face of the mesh. We will use a transformation matrices for that, but before we learn about this topic, we need to start with translations and then proceed to rotations.




```
# cube.obj
#

g cube

v  0.0  0.0  0.0
v  0.0  0.0  1.0
v  0.0  1.0  0.0
v  0.0  1.0  1.0
v  1.0  0.0  0.0
v  1.0  0.0  1.0
v  1.0  1.0  0.0
v  1.0  1.0  1.0

f  1//2  7//2  5//2
f  1//2  3//2  7//2
f  1//6  4//6  3//6
f  1//6  2//6  4//6
f  3//3  8//3  7//3
f  3//3  4//3  8//3
f  5//5  7//5  8//5
f  5//5  8//5  6//5
f  1//4  5//4  6//4
f  1//4  6//4  2//4
f  2//1  6//1  8//1
f  2//1  8//1  4//1
```

Figure 3: Example of a text representation of the model geometry.

4.2.1 Translation

Our models are not stationary and we will want to move them around in our virtual environment. Translation will accomplish just that.

Consider having a 3D triangle:

$$((x_1, y_1, z_1), (x_2, y_2, z_2), (x_3, y_3, z_3)) \tag{4}$$

where we have generic constants for its vertices. Let x_c, y_c, z_c be the amount of how much we want to move the triangle on each axis. Then translation would be:

$$\begin{aligned}
(x_1, y_1, z_1) &\mapsto (x_1 + x_c, y_1 + y_c, z_1 + z_c) \\
(x_2, y_2, z_2) &\mapsto (x_2 + x_c, y_2 + y_c, z_2 + z_c) \\
(x_3, y_3, z_3) &\mapsto (x_3 + x_c, y_3 + y_c, z_3 + z_c)
\end{aligned} \tag{5}$$

where the left side is replaced by the right side after we apply the transformation. If we do this for every triangle of the model, we apply a translation to the whole model.

4.2.2 Rotation in 2D

We call a change of orientation of the model a rotation. This transformation, especially in 3D, is a bit more difficult than translation.

The most convenient form of working with rotations and transformations overall is by constructing a transformation matrix and multiplying every point of the model by such a matrix. Before we dive into this topic, we need to learn about 2D rotation and then we will add the third dimension. We can start by expressing the multiplication of a 2D vector and a matrix.

Consider a two-dimensional general matrix:

$$M = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \tag{6}$$

which we want to apply to the point A represented by coordinates (x_1, y_1). We will notate it like this:

$$\begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} x'_1 \\ y'_1 \end{bmatrix} \tag{7}$$

Using linear algebra, the multiplication yields:

$$\begin{aligned} x'_1 &= m_{11}x_1 + m_{12}y_1 \\ y'_1 &= m_{21}x_1 + m_{22}y_1 \end{aligned} \tag{8}$$

For our multiplication, we will use the notation (x, y) -> (x', y'). A lot of various matrices exist, each doing something different. Figure 4 shows some of the basic transformations. If we use an identity matrix for our transformation, then (x, y) = (x', y'). In case of every position on the diagonal having the same number (noted as k below), but different than number 1 and the rest being zeros, the matrix will cause proportional scaling, (x, y) -> $(k \cdot x, k \cdot y)$. If the numbers on the diagonal differ, it will cause stretching, which changes the aspect ratio. Other matrices may produce shearing or mirror images.

When we want to produce a rotation matrix, we need to be careful not to distort our model. We have to avoid any shearing, mirroring, and changing the aspect ratio to keep our model the same. If we satisfy those rules, it will result in the rotation matrix.

After some deduction [7], the notation of the rotation matrix becomes:

$$R = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} \tag{9}$$

This is the case for model rotation counter clockwise for positive θ. If we wanted a rotation matrix for clockwise rotation of models, we would switch the sign of the sinus parts.

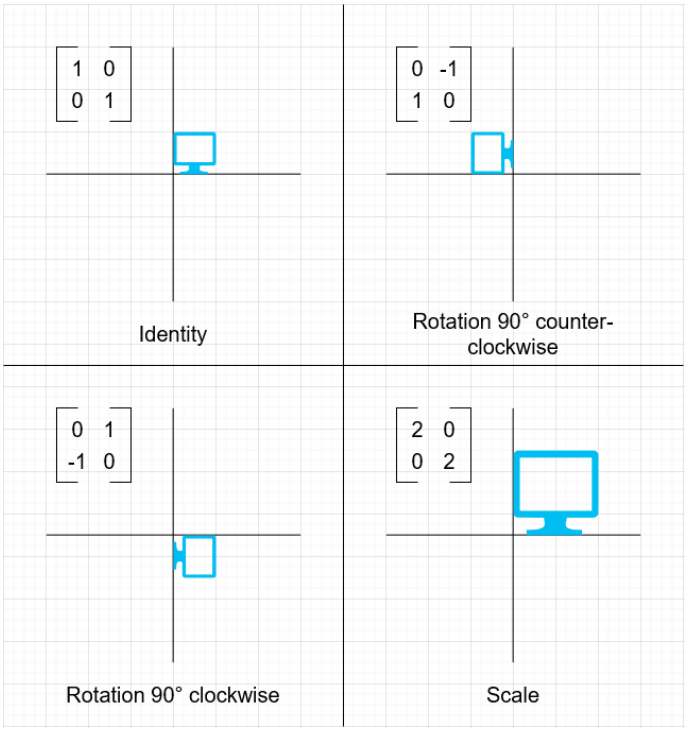


Figure 4: Basic transformation matrices.

4.2.3 Rotation in 3D

At first, we will extend the matrix from Equation 6 and add the third dimension:

$$M = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \tag{10}$$

The easiest way we can accomplish rotations is to divide the complex rotation by individual axes and rotate around each axis individually. Each of the positions on the diagonal are assigned a certain axis, for example the first row and first column would be assigned to an x-axis. If we make every member of the first row and first column zero except for the member on the diagonal, we will find rotation around that axis in the rest of the matrix. In the following examples, you can see this process respectively for x, y and z axes:

$$R_x(\gamma) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\gamma & -sin\gamma \\ 0 & sin\gamma & cos\gamma \end{bmatrix} \tag{11}$$

$$R_y(\gamma) = \begin{bmatrix} cos\gamma & 0 & sin\gamma \\ 0 & 1 & 0 \\ -sin\gamma & 0 & cos\gamma \end{bmatrix} \tag{12}$$

$$R_z(\gamma) = \begin{bmatrix} cos\gamma & -sin\gamma & 0 \\ sin\gamma & cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{13}$$

4.2.4 Transformation matrix

If we wanted to apply rotation and translation to our model with previous knowledge, it would look like this:

$$\begin{bmatrix} & & \\ & R & \\ & & \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} \tag{14}$$

where the R matrix is our 3D rotation matrix, the (x, y, z) vector is the point of our model we want to transform by rotation and the vector (x_c, y_c, z_c) is the translation. If we wanted to chain our transformations, the complexity of the calculation would grow significantly. We would like to avoid this issue and that is where homogeneous coordinates are useful.

If we virtually increase the dimension of our rotation matrix by one, we can fit the translation into the same matrix as well. The best way to show it is on an example:

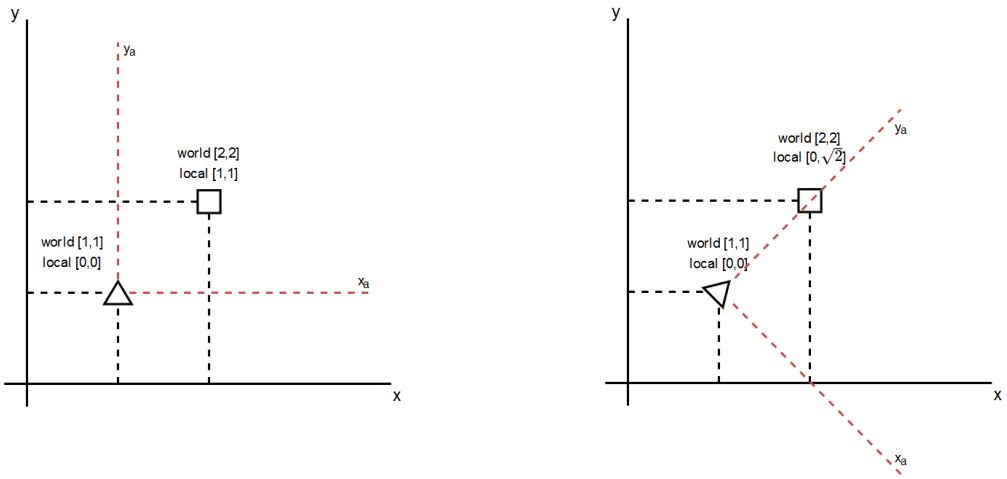
$$T_h = \begin{bmatrix} & & & x_c \\ & R & & y_c \\ & & & z_c \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{15}$$

In this example, we can see the 3x3 rotation matrix noted as R, the translation vector (x_c, y_c, z_c) in the first three positions of the last column and a set of numbers on the last row. Number 1 in the last position is a homogeneous coordinate, which is usually noted as w. We can choose any number we want, but if we choose any other number than 1, we have to multiply the member on each position of the matrix by that number. For example, if we chose our homogeneous coordinate to be $w = 3$, the matrix stated before would look like this:

$$T_h = \begin{bmatrix} & & & 3x_c \\ & 3R & & 3y_c \\ & & & 3z_c \\ 0 & 0 & 0 & 3 \end{bmatrix} \tag{16}$$

If we wanted to go back from homogeneous 4D space back into 3D, we would divide each member of the $4x4$ matrix by the homogeneous coordinate and then split the matrix into our $3x3$ rotation matrix and a translation vector.

When we use homogeneous matrices, we can fit both the translation and the rotation into one matrix. Matrix multiplication is a way easier option of working with transformations than



(a) same orientation

(b) different orientation

Figure 5: Two examples of local and world coordinate spaces - world coordinate system marked in black and local system marked in red. (a) Triangle representing local system position and orientation is oriented in concordance to a world system. (b) Local space is rotated 45 degrees clockwise.

the previous computation, as in Equation 14. In the case of additional transformations, we can just multiply all transformation matrices first and then multiply the result with all of the points of our model. It saves computing power and can result in a smoother and faster transformations. In VR, we need to work in real-time, so this reduction can be important.

4.3 Coordinate spaces

If we look around us, we can see a lot of objects. Every object has its set position in the world relative to us. But what if we want to exactly define its position so the machine understands it? GPS is an example of a coordinate space in our world with its origin set somewhere in the Gulf of Guinea and any position on our globe can be represented as a pair of coordinates describing the distance from this origin. That is an example of a world coordinate space. If we took for example our home as an origin of a different coordinate space, then our home would be at coordinates $[0, 0]$. Every other object would have a coordinate pair relative to our home. This is an example of a local coordinate system. That means every object can have local and world coordinates, depending on what we set as our zero point. A simple example in 2D space is shown in Figure 5. Pay attention to the local coordinates of the square. They changed in the second example because the whole local coordinate system rotated 45 degrees.

5 OpenXR Standard

5.1 Alternative software before OpenXR

No option of user interface (UI) in VR for Blender software was available for a long time. In the last few years, people from MARUI-plugin have been developing this feature as a separate solution until an official one would be done. It is called BlenderXR [13] and I used it as my building stone, because the general solution was not ready when I started developing the add-on. It allowed me not to worry about the headset, because BlenderXR displays correct images in the headset and I could focus on controlling the VR world instead. BlenderXR has an issue with the fact that every VR system has to have a different library for its compatibility with the plugin. For example, HTC Vive and Oculus rift do not work without two different libraries to get their output data.

5.2 Introduction to OpenXR

The general solution is named OpenXR and it should solve this kind of fragmentation. OpenXR specification version 1.0 was released on the 29th of July in 2019 [3]. It is an Application Programming Interface (API) for XR applications, which are any applications running in a real- and virtual-combined environment. It also provides a runtime for communication between the device and the software, such as peripheral management or raw tracking information. It was not specifically developed for Blender but it is available for any software that needs access to AR, MR or VR. The standard is open and royalty-free. The shortcut XR is a broader term containing all of the virtual systems, therefore including AR, MR, and VR applications and systems.

With OpenXR, programmers get a basic set of functions that perform common tasks. It works for every device which implements the OpenXR API. Any company that is developing systems eligible for XR has to implement the API if they want to be a part of the standard. Programmers have to provide a library that implements OpenXR API and they have to map functions of the OpenXR to the hardware. The specification does not provide a model for implementation. Implementation of the runtime just has to produce expected results.

5.3 Goal of OpenXR

The goal of the OpenXR specification is to remove fragmentation and provide a solution, where one code would run on all of the devices without the need of replacing any of its segments. In practice, it would mean that one programmer might, for example, develop a game for an HTC Vive device and it would be possible to play the same game with the same code on Oculus Rift as well.

One of the positives of OpenXR is that it allows software developers to decide on which devices their software will run. They can make their software available for everyone or re-

strict which hardware it will be compatible with. With this approach, it is possible to develop both made-to-measure software and public software for broader use. OpenXR thus gives the developers more freedom to choose which solution fits their software better.

In the time of writing of this thesis OpenXR is still new and it is not so popular, but we might see a broader use of this standard in the upcoming years.

6 OpenVR Interface

Valve Corporation (shortened as just Valve) developed a software development kit (SDK) and an API to support SteamVR [14] called OpenVR [4]. It is an interface between HTC Vive hardware and software I use in this work. The SDK supports controllers of other companies as well, such as Oculus Rift or Windows MR. The API is a set of C++ classes consisting of pure virtual functions. We will use a python wrapper of the OpenVR called pyopenvr in the implementation part of the thesis to provide an easier communication of OpenVR with python. The correct interface is returned after application initialization that complies with the SDK header of the application. Any released version of an interface will be supported in the future and thus the application will not need an update of its SDK when implementing new hardware for the application.

Using this API, we can interact with VR displays and we do not need to rely on the hardware developer to supply us with his SDK. It is possible to access the positions and orientations of the headset and controllers by only one API call and it works the same for all of the brands stated above.

OpenVR consists of two layers: application and driver. OpenVR for applications communicates with SteamVR, which then communicates with an OpenVR driver. Valve does not allow access to SteamVR, but we can develop either an OpenVR application or a driver. OpenVR applications work with SteamVR, access its data like position and orientation of VR hardware, do some operations or display images to the headset displays. An OpenVR driver is a piece of code that integrates a new device into the SteamVR system. This part is not the target of this thesis, so in further sections we will focus on OpenVR applications. The API is divided into seven interfaces:

1. IVRSystem - main interface for display, distortion, tracking, controller access, and event access.
2. IVRChaperone - access to soft and hard bounds of the chaperone.
3. IVRCompositor - 3D content rendering through the VR compositor.
4. IVROverlay - 2D content rendering through the VR compositor.
5. IVRRenderModels - model rendering.
6. IVRScreenshots - request and submit screenshots.
7. IVRInput - define and query invokable actions - creating, editing, and sharing of custom bindings with supported devices.

In the implementation part of this thesis (see Section 8.6), we will come back to this topic and discuss it in more detail.

7 Blender Environment

We will discuss some features that are connected to the Blender [1] environment as a whole. At first, let's have a look at what the individual parts of Blender are. After starting Blender we can see a similar environment to the one visible in Figure 6. The main part of the screen is occupied by different types of editors. After the start, there is one dominant editor called 3D view, but we can switch between the editor types with the button in the top left corner of the editor. It is also possible to add new editors or remove unwanted ones by clicking the corner of the editor when the cursor turns into a cross and dragging it in the desired direction. In the editors we can toggle side menus to access more tools for working with the editor. There is a menu at the top of each editor responsible for working with the specific editor and then a global menu at the top of the application responsible for a general control like creating new projects or changing options of Blender. On the right we can see a list of data-blocks currently used in the 3D view and a properties panel that edits properties of the currently active (selected) object. We can navigate in the editors with the mouse or shortcuts.

7.1 3D View

Inside of the 3D view editor, we work with the scene and the objects inside. We can add new objects including lights for lighting in the scene or cameras for animation and rendering. It is possible to switch between object mode, which works with the whole scene and edit mode, which allows to modify a specific object. It allows us for example to change geometry or prepare the model for textures. In object mode, it is possible to transform the objects, which we will later do in our add-on. In the right sidebar we can see the transformation data about the currently selected object and some information about the scene. It is also possible to change the values there if we want to be more precise and do not want to count on precision in 3D view.

7.2 Shader Editor

Every surface has some properties, e.g. the mirror surface reflects incoming light, the surface of the concrete is rough, each surface has its color, etc. There are different nodes in Blender that represent such properties and by combining them we can imitate real-life materials. Each node has some inputs and outputs (except for the final output node) which we can bind together and create a more complex material. For example if we do not add a specular to a marble material, it will look rather like paint.

After we are done with sculpting the object and assigning the materials to its surface, we might want to get a final image, which is obtained by rendering the scene. The Shader Editor is responsible for editing these materials used for rendering. We can control the Editor the same way as in the 3D View. The editor also contains both sidebars. The right sidebar displays

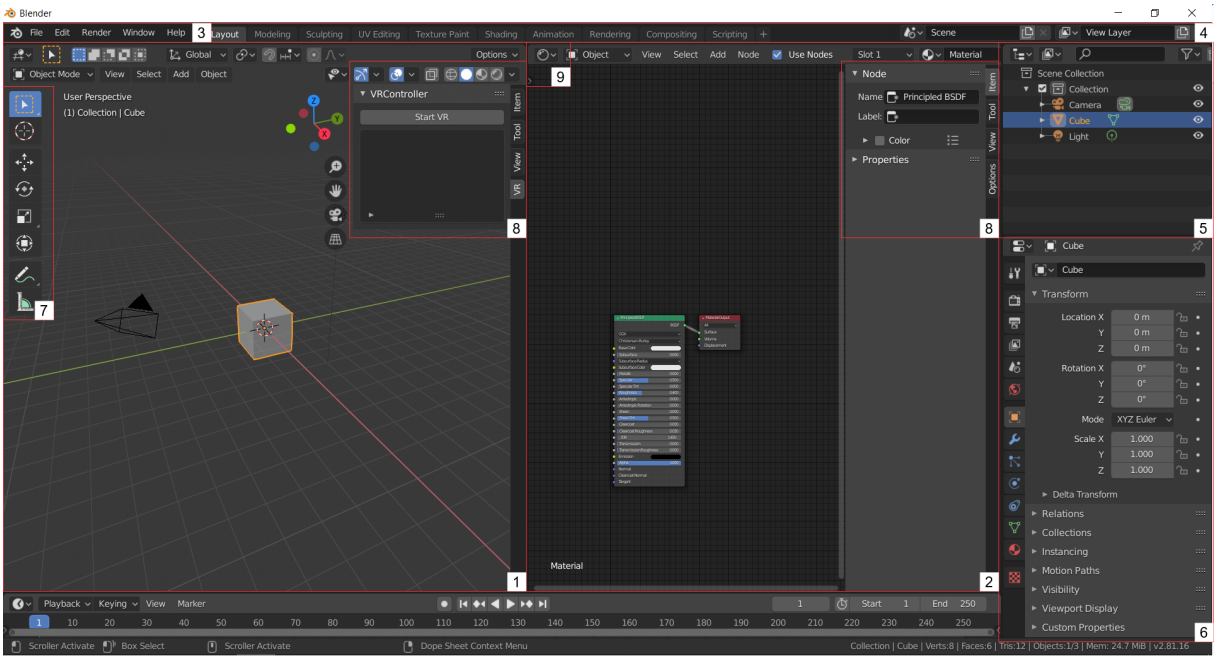


Figure 6: Blender Environment - 1) Main 3D View, where the scene is visible and the only editor which is displayed in the headset. 2) Shader Editor used for working with the appearance of a model. 3) Main menu. 4) Layouts which open certain windows according to their definition. 5) Outliner - shows what items are present in the scene and what data are available. 6) Properties - edits properties of the active object and data related to it. 7) Toolbar - works with objects in the scene. (T shortcut) 8) Sidebar - Object 3D properties (position, rotation), menu where our add-on is located. (N shortcut) 9) Button for switching between editors.

data of the currently selected node the same way it would be visible in the editor, only without confusion about whether we are looking at the correct node.

8 Implementation

In my implementation part of the thesis, I will explain my version of controlling Blender in VR. OpenXR standard 1.0 [3] came out after I started working on my thesis, so the first step for me was to research if there were already any functional solutions and I decided to use BlenderXR [13] as my starting point. Because the goal was to develop only the controlling part of the VR and not the displaying one, I used their code to create a new window in Blender specifically designed for VR and used their way of displaying the content from Blender in the VR lenses. BlenderXR has already developed the operation of controllers, but I substituted its code with mine and used my approach, because its approach modified the UI, which was not desirable. My goal was to develop the control in such a way that it would be fully functional with the original UI that was already present in Blender.

8.1 Features

My add-on adds features to the Blender that allow us to control its environment while in VR. When we open a new VR window in BlenderXR, only 3D View (see Section 7.1) is visible. Some features are addressing that issue, because we want to control the whole environment, not just 3D View. After allowing the add-on in preferences, a new item in the right menu of the 3D view appears. We can click on the start button and after the add-on is ready and it imports all the objects that represent individual parts of the VR system (see Section 8.4), these actions become available:

- Move - Controllers are responsive and they move according to their physical position (see Section 8.6).
- Select objects - With the right controller, we can point at an object and select it (see Section 8.6).
- Move objects - After an object is selected, we can move it in the scene.
- Open menu - Menu button on the left controller opens a projection plane with the whole Blender environment (see Section 8.6.2).
- Work with the environment - The right controller emulates mouse actions on the plane. It is possible to select items from menus, change properties of objects in the scene, control different editors, and customize the content on the plane (see Section 8.6.2).

8.2 Application development approach

Blender is highly customizable and anyone can create new modules that modify the behavior of Blender and help the users to work more efficiently. Such a module is called an add-on. There is an extension for Visual Studio Code [15] that helps with coding in Blender. It can execute

scripts or create new add-ons. For a new add-on, it creates a folder with a python file that contains a basic template for a functional add-on. If we install a finished add-on in Blender, we can just enable it in the preferences and start using it right away.

Every add-on created as a python module needs to contain three basic parts. A `bl_info` variable, which is an array with metadata and two functions `register` and `unregister` that are responsible for enabling and disabling of the add-on when working with the result in the preferences of Blender. We can see an example of a simple add-on structure in Listing 1.

```
bl_info = {
    "name" : "Add-on",
    "author" : "Martin Sovjak",
    "description" : "This is an add-on",
    "blender" : (2, 80, 0),
    "version" : (0, 0, 1),
    "location" : "View3D > UI > View > VRController",
    "category" : "3D View"
}

def register():
    auto_load.register()

def unregister():
    auto_load.unregister()
```

Listing 1: Add-on basic example.

Most of the variables in the `bl_info`(Listing 1) are useful to describe basic information about the add-on and are visible in the preferences of the UI. Location tells us where we can find functionality of the add-on. Category is used for filtering. One add-on can belong only to one group (for example 3D View). Listing 1 contains only some of the most important fields, but the list is not restricted to these options only.

It was necessary to add some UI to my add-on to be able to control its behavior. I created a new class for a panel, that tracks how many VR devices are connected. I added a class that is responsible for adding a sidebar into the Blender UI with a start (or stop) button and the panel for tracking. In the `register` function, we can add our classes and modify our Blender UI. An example of registering a panel is shown in Listing 2.

```
def register():
    bpy.utils.register_class(VIEW_PT_VRPanel)
```

Listing 2: Example of registering a class.

8.3 Handlers

My add-on has to be capable of refreshing 3D view at a frame rate comfortable for a human eye, which is usually at least 24 frames per second, but for VR at least 60 frames per second are optimal. I wanted an easy and convenient solution, so I used a Blender feature that allows binding handlers onto some of its available events. My event of choice was a change of frames in the animation panel.

```
    bpy.app.handlers.frame_change_pre.append(changeFrame)
```

Listing 3: Example of binding a handler onto an event.

When we are creating a new handler 3, we have to pass a function as a parameter. In my case, I have a class that contains a method dedicated to loop every frame. It is responsible for checking for VR input and changing of what is being displayed on the screen. It is not possible to bind a class or a class method to a handler, so I made a separated function called `changeFrame` whose only task was to call the loop method. I passed it as a parameter to the method that created the handler.

After pressing space on a keyboard to start an animation (or starting an animation manually), before every frame change, the method is invoked and respective actions within that method are taken.

8.4 Object Preparation

We need to be able to see something in our VR headset to have a successfully functioning add-on. Users need to see, where they are moving controllers with their hands. Steam has its models of the Vive controller, headset, and Vive lighthouse in a Steam folder with an appropriate `.obj` suffix which we can import into our Blender application. We need to import two controllers, one for each controller users have to hold. Then a plane is created for displaying menus and overall control. For better orientation, two lighthouse objects are imported in the scene and a model to represent the headset. The last model is a little ball substituting for a mouse when trying to navigate in the Blender environment in VR. In Listing 4 we can see an example of creating a lighthouse object. I call a Blender operator for importing objects and then I modify some of the parameters of the imported object, such as dimensions and its name. In Figure 7 we can see all the models imported to Blender by my add-on.

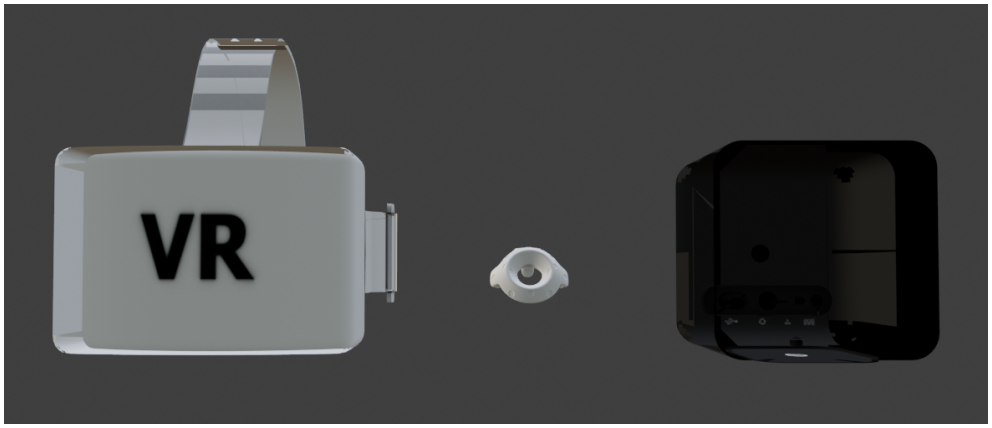


Figure 7: Imported models.

```
file_loc = "./lh_basestation_vive/lh_basestation_vive.obj"
bpy.ops.import_scene.obj(filepath = file_loc)
back_lighthouse = bpy.context.selected_objects[0]
back_lighthouse.name = "back_lighthouse"
back_lighthouse.dimensions = (0.3,0.3,0.3)
bpy.ops.object.transform_apply(location=False, rotation=False, scale=True)
```

Listing 4: Importing an object.

8.5 Blender OpenGL Wrapper

OpenGL [16] is an open-source library with an API for rendering 2D and 3D vector graphics. It is designed to interact with the graphics processing unit (GPU). It can be run entirely on software, but running it on hardware achieves faster and higher-quality rendering. The final image is acquired by a rendering pipeline, where on one side there is the data about the final image and the result is on the other side. The data pass through a chain of components that do specific actions, for example a Vertex Shader, which is responsible for working with the geometry of our models (but not modifying the geometry itself).

There is a Python wrapper for OpenGL in Blender called bgl [17], which implements API calls to make the functions available for Blender. Normal Blender UI is not visible in the VR window displayed in the headset. Displaying Blender UI on a plane object, which is visible in a VR window, is a viable option. We can utilize bgl to achieve such a result. OpenGL allows mapping a texture, which is essentially a 2D image, with a texture unit onto a desired model. I made a texture from the whole Blender window and I displayed it on a plane. Whenever a menu button on the left controller is pressed, it toggles the visibility of the plane and moves it to the current position of the controller. This way the plane can serve as a menu. How the texture is created is shown in Listing 5.

```
img = bpy.data.images.get("texture")
img.gl_load()

tex = img.bindcode

glBindTexture(GL_TEXTURE_2D, tex)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, 0)

glCopyTexImage2D(
    GL_TEXTURE_2D, #target
    0, #level
    GL_RGBA, #internalformat
    0, #x
    0, #y
    self.scene.interactive_plane.screen.width, #width
    self.area.height, #height
    0 #border )
```

Listing 5: Creating a texture out of the Blender window.

We need to somehow bind the texture to the plane object. The object holds data called materials that contain information about its qualities and its different properties (e.g. color or roughness). In Blender, we can bind an image to a material and assign that material to an object. I created an image with the size of the screen that was supposed to be displayed. Then I added a new material to the plane object, bound the image to the material, and adjusted the color and brightness of the material for better contrast, when the final image is displayed. This has to be done before the code in Listing (5) is executed. When executed, the texture is bound and the parameters in the following lines configure the currently bound texture. `GlTexParametri` commands define how the texture should be displayed. The first two commands ensure the transitions between colors will be smooth, the second two disable unnecessary stretching. After that, `glCopyTexImage2D` copies pixels into a 2D texture image and the original image I have created is replaced by the texture. We need to pass some arguments to the `GlCopyTexImage2D`. It needs the memory location where it should create the texture, which is the previously created placeholder image, then it needs to know from what pixel it should start copying and the size

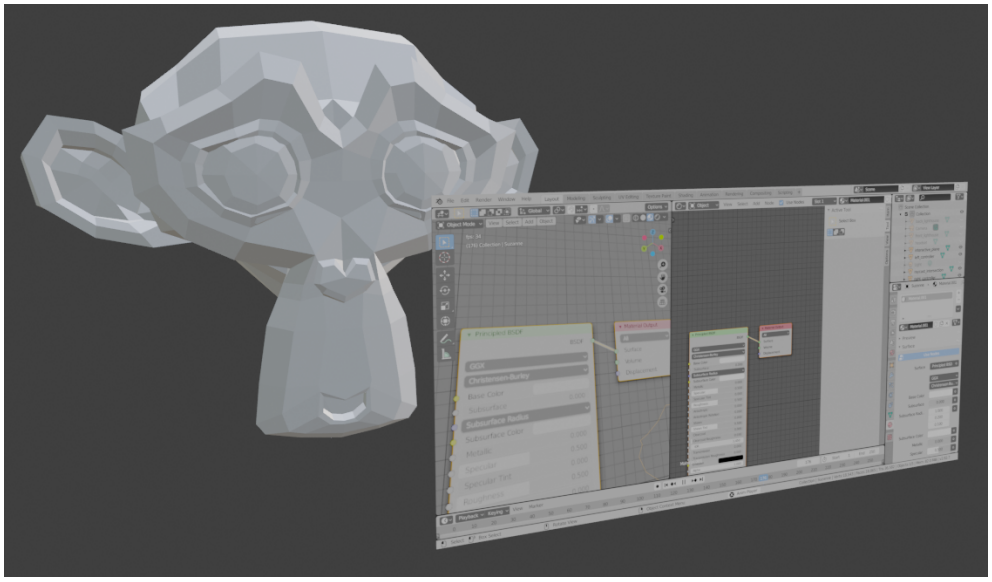


Figure 8: Projection plane used as a menu to control the Blender environment in VR.

of the copy in pixels. Because the placeholder image is bound to a material belonging to the projection plane, the copied window is visible on it right away after this code is executed. The final result is shown in Figure 8.

8.6 Obtaining VR Data

Every device of the HTC Vive system communicates with the PC through SteamVR runtime. We can use SteamVR API to obtain data that are returned from the devices. As stated in the Section 6, there are seven interfaces of the API. I am using three of those interfaces, which are `IVRSystem`, `IVRCompositor`, and `IVRInput`. Every interface is represented by a class in pyopenvr and contains a set of methods that can be invoked.

8.6.1 IVRSystem interface

`IVRSystem` interface is dedicated to obtaining tracking information. It contains methods for getting information about the headset, controllers, and events that are currently happening. Most of the interface only returns output information from the devices and does not send anything back. In my implementation, this interface is the most crucial and I will describe some of the most important methods used in the code. When we get an array of currently tracked devices (see Section 8.6.3) (controllers, HMD or lighthouses), we can call `getTrackedDeviceClass` with an index of the device retrieved from the array and get information about what type of device is at that position. This is necessary for differentiating between controllers and other device types. Using this method, I iterate through the whole array and filter devices I will use into separate arrays. After I get the arrays, I want to obtain tracking information about the devices. Every device is represented by an object. One of its class member fields is called

`mDeviceToAbsoluteTracking`, which is a 3x4 transformation matrix. It is represented differently from Blender matrices, so I had to create a method to rearrange and change its fields into a matrix compatible with Blender. The transformation is shown in Equation 17.

$$\begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ b_0 & b_1 & b_2 & b_3 \\ c_0 & c_1 & c_2 & c_3 \end{bmatrix} \rightarrow \begin{bmatrix} -a_0 & -a_1 & -a_2 & -a_3 \\ c_0 & c_1 & c_2 & c_3 \\ b_0 & b_1 & b_2 & b_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{17}$$

Models in Blender contain information about their position in 3D space in the form of a $4x4$ matrix. After transformation of the matrix obtained from `mDeviceToAbsoluteTracking`, it is possible to assign the matrix to a corresponding Blender model. Matrix assignment is shown in Listing 6.

```
our_desired_blender_model.matrix_world = transformed_matrix
```

Listing 6: Example of matrix assignment.

The next important method of the `IVRSystem` interface I use is `pollNextEvent`. This method is used to get information about the events fired at the time of the call. This allows me to loop through all of the events and check which device fired the event. If the event is fired by a controller, we can process it and resolve what should happen with the environment (see Section 8.6.2). It is very useful to be able to communicate with the Blender environment as a user, because this way the add-on is interactive and we can use controllers to operate in the environment.

8.6.2 Events and controller bindings

Any action our VR system takes has its structure in OpenVR called `VREvent`. Because I am using the pyopenvr wrapper, it is represented by an object instead. Each event has a structure of `VREvent_*nameOfEvent*`. It consists of its type, index of the device that fired the event, its age, and additional data. Additional data is a union of many structs and provides more information about the event. If we want to process controller specific events, we can find them under the field `event.data.controller.button`. The name of such a controller event is composed like this `k_EButton_*nameOfButton*`.

When processing a button press of a controller, a chain of events is fired. The sequence is as stated:

1. VREvent_ButtonTouch
2. VREvent_ButtonPress
3. VREvent_ButtonUnpress

4. VREvent_ButtonUntouch

In the case of the trigger button, it is possible to fire only `ButtonTouch` and `ButtonUntouch` events if only light pressure is applied to it. Because of its inconsistency I do not check for `ButtonTouch` and I use only the `ButtonPress` and `ButtonUnpress` sequence. When a `ButtonPress` event is fired, I check the previously mentioned button field for information which button was pressed. There is an option for every button on the controller (see Figure 2 for different button types), therefore the list consists of these options (stated with a description which button it corresponds to):

1. k_EButton_System - Menu button above the trackpad.
2. k_EButton_ApplicationMenu - Menu button below the trackpad.
3. k_EButton_Grip - Button on either side of the controller.
4. k_EButton_SteamVR_Touchpad - Trackpad.
5. k_EButton_SteamVR_Trigger - Trigger button located on the back side of the controller.

I can decide what should happen based on this information and the device index field that gives me data about whether the left or right controller was pressed. For each controller, I have a set of `if` statements that filter actions of the controller according to the list above after every `ButtonPress` event. The right controller is used as a main source of control, the left one is secondary. Concerning the events of the right controller, the trigger event is responsible for the emulation of the left mouse button. When we press it lightly, it performs a simple click, if we press it for a longer time, it works as a selection tool in the same way the mouse would. Here is a description of the individual responsibilities of buttons on the right controller:

- Trigger button - Emulates left mouse button click.
- Grip button - Emulates right mouse button click.
- Trackpad - Zoom, acts like a scroll wheel.

and here is the same description for the left controller:

- Menu button - Switch between the scene selection mode and the menu selection mode (the one with the projection plane).
- Trackpad - Changes size of the projection plane.

8.6.3 IVRCompositor

`IVRCompositor` interface takes care of distortion, prediction, and synchronization necessary for a good experience in VR. We can use `waitGetPoses` to get the list of currently connected devices to SteamVR. The interface then allows us to render the left and right eye of the headset and then display both of them in its window. We only need the `waitGetPoses` method of the interface. We store the list that is returned by the method and use it with `IVRSystem` to obtain all the information we need from the connected devices, such as tracking data and events.

8.7 Pynput

Pynput [18] is a python library providing keyboard and mouse emulation and allows us to process events these peripherals produce. It is implemented as an almost unified API for every supported platform (macOS, Linux, or Windows). Whenever a user has the projection plane opened and wants for example to select an item from a menu or move a node to a different location, this information is processed and then the mouse performs the desired action.

Pynput mouse control is located in the `pynput.mouse.Controller` part of the library. The initialization is shown in Listing 7.

```
from pynput.mouse import Button, Controller

mouse = Controller()
```

Listing 7: Pynput initialization.

After initialization, we can call methods on the mouse object or set its variables. In Listing 8 there is an example of setting the position of the mouse, performing left and right clicks and using the scroll wheel:

```
mouse.position = (10, 10)
mouse.press(Button.left)
mouse.release(Button.left)
mouse.press(Button.right)
mouse.release(Button.right)
mouse.scroll(10,20)
```

Listing 8: Working with Pynput mouse object.

In the case of setting the position of the mouse or scrolling the wheel, we just need to pass a tuple with two coordinates in pixels, the axes being X and Y respectively. For every press method called, it is necessary to pair it with a release method, otherwise it would act the same

way as if the mouse button was still held down. In the Listing 9 there are the last two methods which are not used in my implementation, but might be useful:

```
mouse.click(Button.left, 2)
mouse.move(x, y)
```

Listing 9: Other useful methods. The first method performs a defined number of clicks(second parameter) with a specific button(first parameter). The second method moves the mouse by x pixels on the X-axis and by y pixels on the Y-axis.

The first method performs a double click with the left mouse button. The number of clicks is decided by the second passed parameter and the button is decided by the first parameter. The second method moves the mouse by x pixels on the X-axis and by y pixels on the Y-axis.

8.8 Object Selection and Movement

One of the features Blender Python API contains is a method that decides if a ray cast from a point in a desired direction intersects with an object. This method is called `ray_cast` and is implemented in two different ways. The options are either casting the ray (thus the action is called raycasting) in the world space or the local space (see Section 4.3).

In the case of the world space ray-cast, the ray is cast from a defined position and if it intersects with any object, the closest one will be returned with additional data about the intersection. The additional data include a location vector of the intersection, the face normal vector of the face that was hit, the face index, and the transformation matrix of the intersection. The method takes four parameters with the last one being optional. The method definition is shown in the Listing 10.

```
scene.ray_cast(view_layer, origin, direction, distance = 1.70141e+38)
```

Listing 10: Scene ray-cast definition.

`View_layer` stands for a scene layer, where the ray should be cast. `Origin` and `direction` are both three-component vectors. Both vectors are in a world space and their names are self-explanatory. `Distance` is an optional parameter defining the length of the ray. If it is not provided, the maximum distance is used instead. This method is very useful when working with many objects in the scene, for example when selecting objects with the controllers in VR.

Unlike the scene ray-cast, object ray-cast works with a specific object instead of the whole scene. It returns the face index, normal of the intersection, and the location of the ray-cast hit location. The method takes four arguments as well, but only two of them are required.

```
object.ray_cast(origin, direction, distance = 1.70141e+38, depsgraph = None)
```

Listing 11: Object ray-cast definition

Both `origin` and `direction` parameters work similarly to the scene ray-cast, but with the exception that both vectors have to be defined in a local space, not the world space. Vectors are by default in the world space, so we have to do some transformations to obtain a vector in the local space (see Section 8.9). The distance parameter is the same as in the previous method. Depsgraph stands for the dependency graph and should be supplied only if the current one in `Context` is not suitable. The dependency graph is responsible for dynamic updates in the scene but it is not necessary to supply it in my implementation. The method returns ray-cast hit location as a three-dimensional vector in local space, where the possible values of the vector are dependent on the object origin. Origin in the middle of the model would result in values ranging from $[-\frac{1}{2}x, -\frac{1}{2}y, -\frac{1}{2}z]$ to $[\frac{1}{2}x, \frac{1}{2}y, \frac{1}{2}z]$ where x, y, and z correspond to the width, height, and depth of the object. I am using this method when selecting from the plane on which I project Blender environment. I do not need to take the Z-axis into account, because the plane has no depth. It allows me to precisely know where the ray coming from the position of the controller hits the plane. If I use information about the location as parameters for `loc` variable in the Listing 12, I can move the mouse to the desired location.

```
hit_x_recalculated = (loc[0] + self.initial_width / 2) / self.initial_width
mouse_x_position = hit_x_recalculated * self.screen.width + self.screen.x

# loc[0] - location of a ray-cast hit in meters, x coordinate
# self.initial_width - original width of the projection plane in meters
# self.screen.width - width of the screen projected onto the plane in pixels
# self.screen.x - the distance from the left edge of the computer screen to the
    projected area

hit_y_recalculated = (loc[1] + self.initial_height / 2) / self.initial_height
mouse_y_position = self.screen.height - (hit_y_recalculated * self.screen.
    height) + 20

# loc[1] - location of a ray-cast hit in meters, y coordinate
# self.initial_height - the height of the projection plane in meters
# self.screen.height - the height of the screen projected onto the plane in
    pixels
```

Listing 12: Calculation of mouse position from object ray-cast location result.

My plane has its origin in the middle. I need values in a range of $< 0, 1 >$, so I add half of the width (height respectively) to the ray-cast location results. In the case of mouse x coordinate, I need to multiply this value with the pixel width of the area projected onto the plane and then add the distance between the left window edge and the projected area. When calculating y coordinate, we need to do almost the same thing, but then subtract the result from the height of the window, because the y coordinate is inverted with zero being in the top left corner of the screen instead of the pynput's bottom left corner. To get the correct position of the mouse, we need to add twenty pixels in the end, because `window_height` does not take Blender navigation bar into account and pynput does.

8.9 Vector transformations

Getting the right vectors for the `raycast` methods is a little bit difficult. We have already gone through the difference between local and global coordinate spaces (see Section 4.3). To understand the next section, we need to introduce an option that would allow us to transform location of a point represented by a vector in one coordinate system to the location represented in a different coordinate system. To achieve this, we need a transformation matrix which is provided by expressing the transformation between the respective coordinate systems. If we took the Figure 5b, an example of the transformation would take place as follows: Take the vector expressing the position of the square in the local space and then convert it to a description of position in the world space. The transformation is possible both ways, but that is true only because the M_w matrix in Equation 18 is invertible. Also, vectors in Blender are represented as (x, y, z), but the transformation matrix is 4×4. At first, Blender has to extend the vector by the homogeneous coordinate (see Section 4.2.4). The vector then becomes $(x, y, z, 1)$ and it is possible to multiply it by the matrix. The result is again a 4×1 vector, but Blender only saves the three meaningful coordinates. In the Equation 18, we can see the expressions to obtain the vectors in respective spaces (lines 1 and 4). We can use linear algebra operations with matrices to prove conversions between the two coordinate spaces. This proof is also shown in the Equation 18,

$$\begin{aligned} v_w &= \mathbf{M}_w v_l \\ \mathbf{M}_w{}^{-1} v_w &= \mathbf{M}_w{}^{-1} \mathbf{M}_w v_l \\ \mathbf{M}_w{}^{-1} v_w &= \mathbf{I} v_l \\ v_l &= \mathbf{M}_w{}^{-1} v_w \end{aligned} \tag{18}$$

where v_l stands for a vector in the local system, $\mathbf{M}_w$ is a transformation matrix to world space, v_w is a vector in the global (world) system and $\mathbf{M}_w{}^{-1}$ stands for an inverted matrix of the original matrix $\mathbf{M}_w$.

The Equations in 18 are very important in the following listings for transformations between coordinate spaces. In the Listing (13) we can see the preparation of the parameters used in the next two Listings (14 and 15).

```
# Create an initial origin vector, or rather the point from where the ray will
    be cast
origin = mathutils.Vector((0,0,0))

# Create an initial direction vector, it should be facing the projection plane
    and the corresponding axis from the front to the back of the projection
    screen is -z, that is why the vector has -1 in the third column.
direction = mathutils.Vector((0,0,-1))

# Get transformation matrix of the controller with which we select options on
    the projection plane
shooter_mat = controller_matrix

# Get transformation matrix of the projection plane
plane_mat = self.interactive_plane.matrix_world
```

Listing 13: Parameters used in the calculation of the vectors used in `object.ray_cast` method.

In the Listing (14), we can see a part of the method that is responsible for obtaining the origin parameter used in the `object.ray_cast` method.

```
def get_local_matrices(origin, direction, matrix_from, matrix_to):

    # Invert the matrix to get from world to local space
    matrix_to_inv = matrix_to.inverted()

    # We multiply the previously created vector by the controller matrix to
        move the vector from the local space of the controller to the world
        space and then by the inverted projection plane matrix we move it into
        the local space of the projection plane
    point_local = matrix_to_inv @ matrix_from @ origin
```

Listing 14: Transformation of the vector for the origin parameter of the `object.ray_cast` method.

The Listing (15) shows how to obtain the direction parameter used in the `object.ray_cast` method.

```
def get_local_matrices(origin, direction, matrix_from, matrix_to):
    # Copy the matrix of the right controller and the projection plane
    matrix_from_copy = matrix_from.copy()
    matrix_to_copy = matrix_to.copy()

    # invert the projection plane matrix
    matrix_to_copy_inv = matrix_to_copy.inverted()

    # Location (the last column of the matrix) of both matrices is set to the
        world origin
    matrix_from_copy[0][3] = 0
    matrix_from_copy[1][3] = 0
    matrix_from_copy[2][3] = 0

    matrix_to_copy_inv[0][3] = 0
    matrix_to_copy_inv[1][3] = 0
    matrix_to_copy_inv[2][3] = 0

    # Direction is computed by multiplying the two matrices and the direction
        vector
    direction_local = matrix_to_copy_inv @ matrix_from_copy @ direction
```

Listing 15: Transformation of the vector for the direction parameter of the `object.ray_cast` method.

On the last line of Listing (15), direction vector is at first multiplied by the transformation matrix of the right controller to represent the vector in the world space so it would point a one unit (one meter in case of Blender) to the $-z$ axis from the position of the right controller. We need to then represent it in the local space of the projection plane, because the `object.ray_cast` method requires a direction vector in local space as a parameter. This is accomplished by multiplying the previous result by an inverted matrix of the projection plane, because we are transforming from the world space into the local space (see the second part of Equation 18). It is necessary to translate both the right controller and the projection plane matrices to the origin, because otherwise, the direction might be greatly distorted.

I was using the scene ray-cast method as well, which requires the vectors to be in the world space, so if we used just the parts of the listing that are responsible for representing the vector into the world space and omitted the lines that work with the projection plane and multiplication of its matrix with the result in the world space, the process would stay the same.

8.10 GPU shader module

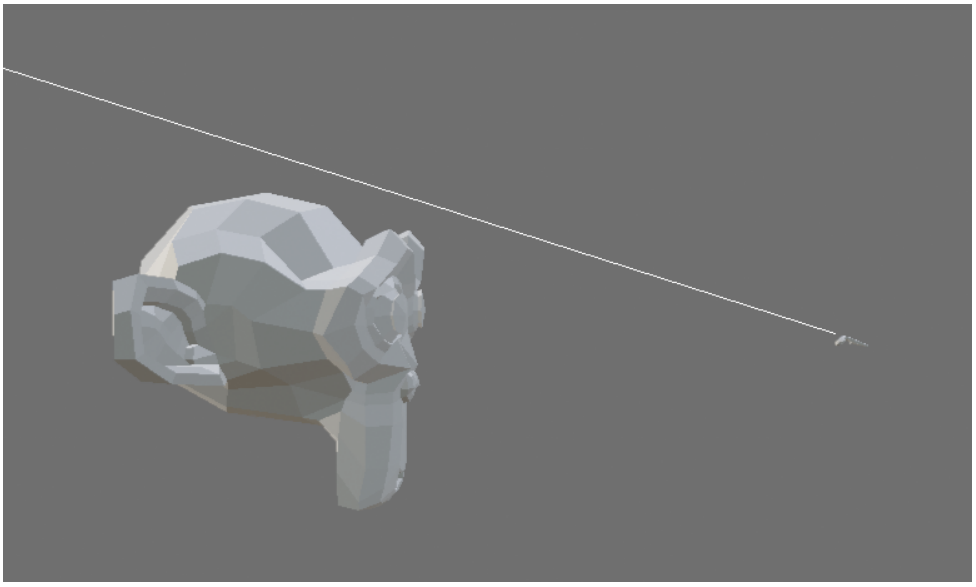
Even though ray-cast methods cast the ray in the Blender environment, it is invisible for the user. The ray should be visible because the user needs to be able to identify what the ray is hitting. In the case of the scene ray-cast, the user needs to know, what object is he trying to select. In the case of the object ray-cast, the user needs to know the location on the projection plane, where the ray hits it, to be able to select anything from a menu or drag something in an editor (e.g. a node inside the Node Editor). Blender gives us an option to use shaders that are used for drawing inside of its 3D view. This feature can be found in the `gpu_extras.batch` library and we will need one of its methods called `batch_for_shader`, because it takes care of all of the necessary attributes needed for a specific shader.

Shader is a computer program used to operate individual parts of GPU. We can give it a set of data, such as vertices or colors, it will process them with GPU and give us an output, in our case a visible line. In Blender, the term Shader refers to an OpenGL program. Each shader consists of a vertex shader, a fragment shader, and a geometry shader. At first, we need to get the correct shader which will output our desired result. A shader that draws single-colored lines is sufficient for us. Then we need to call the method `batch_for_shader`, that will take the shader and some other parameters and create a batch, which contains necessary data for drawing. Other parameters are a vertex array defining the location of the drawing and a type that specifies how the vertices should be processed. We need to draw only one line, so the `'LINES'` type is good enough. This tells the method it should split the vertex array after every two vertices and make a line between each pair. We then need to create a method that will render the batch in the 3D view and add it to a drawing handler which is necessary for the render to be visible. The full code is in the Listing 16 and an example of the result is shown in the Figure 9.

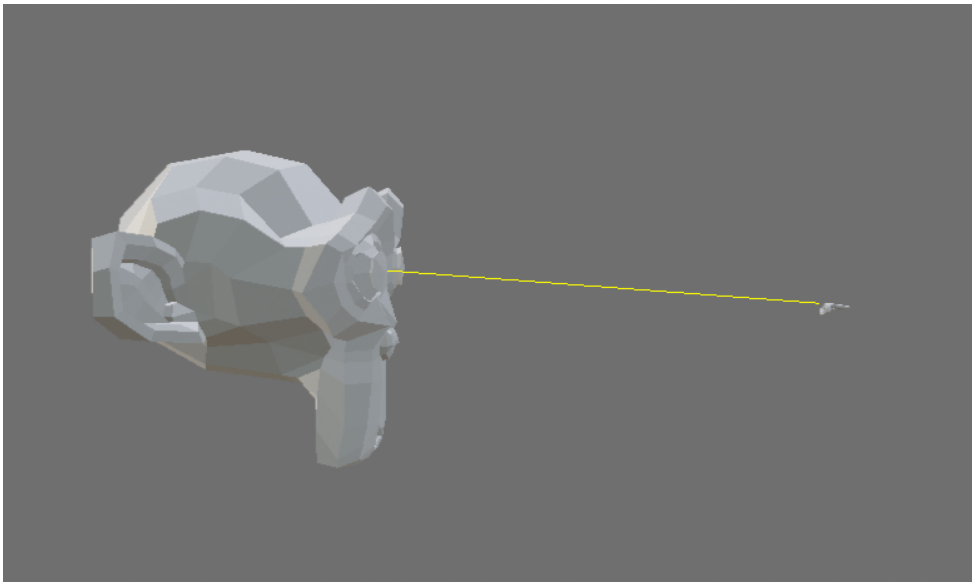
```
# Draw method that will be bound to the handler
def draw(self):

    # Tell which shader should be used for rendering
    self.shader.bind()

    # Set the color of the line depending whether the user is pointing at an
        object or not
    if self.object_in_track:
        self.shader.uniform_float("color", (1, 1, 0, 1))
    else:
        self.shader.uniform_float("color", (1, 1, 1, 1))
```



(a) Controller does not point at the object.



(b) Controller does point at the object.

Figure 9: Lines created with GPU shader.

```
    # Render the batch with the shader
    self.batch.draw(self.shader)

# set a vertex array for the batch, create a shader and compile it to the batch
     for drawing

self.shader = gpu.shader.from_builtin('3D_UNIFORM_COLOR')
self.batch = batch_for_shader(self.shader, 'LINES', {"pos": self.scene.
    get_coords()})

# add the draw method to the draw handlers
self.handle = bpy.types.SpaceView3D.draw_handler_add(self.draw, (), 'WINDOW', '
    POST_VIEW')
```

Listing 16: An example of rendering a ray from raycast into 3D view.

When working with shaders (see Listing 16), we first prepare the vertex array, create a shader, compile it to the batch and bind a draw function to the handler. In the draw method, we just specify what shader should be used (the one we have created or a different one if we did not create any), set other variables such as the color, and then render the batch. The `self.object_in_track` variable is used only when using the `scene.ray_cast` method, so when we are working with the projection plane, the line will never change colors.

8.11 Blender environment adjustment

After Blender is started, a default environment is brought into the screen. We can switch between different predefined setups, but we need to do custom adjustments to the environment for the add-on. The add-on creates a new item in the right menu of 3D view (opened by the "N" key). By default, Viewport Shading (one of the four round icons in the top right corner of the 3D view) is set to solid. After starting the add-on, it is changed to rendered. This allows us to see textures in the scene which allows our objects to be correctly presented in the 3D View. It is also necessary to display the projection plane with the Blender environment, because otherwise it would not be visible.

After all the steps are finished, users can see Blender 3D View in their headset and can control the Blender environment with the controllers. It is possible to move inside of the environment by walking or by turning the head around. We can select any selectable object inside of the scene, move it or rotate it. Unselectable objects are any objects imported by the add-on, which are used for controlling the environment. When an object is selected, we can turn on the projection screen that represents a part of the screen and acts like a menu. We can work with menus on the projection plane the same way we would with the mouse. It is possible to select a single node or perform a box select for a bigger selection. If anything is not visible, we can move in

the editor or use zoom. If the projection plane is too small or too big, we can adjust the size of it as well.

9 Conclusion

As we could see in this work, VR is a large field with a lot of information to learn. It incorporates information from multiple areas such as engineering for the construction of VR systems, physics for working with light and lenses, biology for information about the human body and how it reacts to VR stimuli and mathematics for calculating the projection on the displays and others. VR is developing fast and sees more users every day. Scientists can use it for 3D visualizations of their data which would be too hard to understand otherwise. Students can learn by observing the virtual 3D scene instead of written words. The options are almost endless.

In this thesis, I have tried to outline some basic principles of the previously mentioned fields. We have learned about the individual parts of the VR and how the lenses inside the headset affect incoming light. We have discussed the basics of transformations in 2D and 3D. We have spoken about Blender options for VR, specifically the OpenVR interface and the OpenXR standard. I have described the Blender environment to better understand the implementation.

The main goal of the practical part of this thesis was to create a tool that would provide a way to control Blender and possibly visualize scientific data in VR with the tool. This goal was achieved by using a variety of standard libraries and by modifying an existing solution for VR in Blender for my purposes. The first step was to check the market for available software that might help me. I used BlenderXR to provide a working solution to display content in the glasses. OpenVR with its python wrapper (pyopenvr) provided a reliable way of getting positional data from the HTC Vive system. Standard libraries such as pynput helped me to achieve the functionality I needed. The second step was to create the tool itself, which would be suitable for Blender. I created an add-on that allows the user interaction with the Blender environment in VR. It is possible to extend the add-on by adding more options for the user. Adding more users that could work in the same scene would be also a great addition. The issue with the add-on in the current state is that it relies on BlenderXR. It would be better to switch towards OpenXR standard instead and it is to be expected in the future.

While working on this thesis, I acquired technical knowledge about HTC Vive, Blender, and virtual reality overall. I learned how to create add-ons in Blender, how to work with SteamVR and I learned new information about OpenXR.

References

1. *Blender: Open source 3D creation.* 2020. Available also from: https://www.blender.org/.
2. *HTC Vive*. HTC Corporation, 2020. Available also from: https://www.vive.com/us/.
3. *Khronos Releases OpenXR 1.0 Specification Establishing a Foundation for the AR and VR Ecosystem*. 2019-07. Available also from: https://www.khronos.org/news/press/khronos-releases-openxr-1.0-specification-establishing-a-foundation-for-the-ar-and-vr-ecosystem.
4. *OpenVR* [online]. Valve Corporation, 2020 [visited on 2020-05-04]. Available from: https://github.com/ValveSoftware/openvr.
5. *Microsoft Hololens: Microsoft Hololens* [online]. 2020 [visited on 2020-04-28]. Available from: https://www.microsoft.com/en-us/hololens.
6. *Google glass: Google glass* [online]. 2020 [visited on 2020-04-28]. Available from: https://www.google.com/glass/start.
7. LAVALLE, Steven M. *Virtual Reality* [online]. Cambridge University Press, 2019 [visited on 2020-02-20]. Available from: http://vr.cs.uiuc.edu/vrbook.pdf.
8. EGGER, J.; GALL, M.; WALLNER, J.; BOECHAT, P.; HANN, A.; LI, X.; CHEN, X.; SCHMALSTIEG, D. *HTC Vive MeVisLab integration via OpenVR for medical applications.* [online]. 2017 [visited on 2020-03-15]. Available from: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0173972.
9. YU, F.T.S.; YANG, X. *Introduction to Optical Engineering*. Cambridge University Press, 1997. Introduction to Optical Engineering. ISBN 9780521574938. Available also from: https://books.google.cz/books?id=RYm7WwjsyzkC.
10. *Snell's Law: On the Potential Use of Evolutionary Algorithms for Electro-Optic System Design - Scientific Figure on ResearchGate* [online]. 2019 [visited on 2020-04-28]. Available from: https://www.researchgate.net/figure/Illustration-of-Snells-law-where-n2-n1-The-angle-th1-is-measured-between-the-incident_fig9_235132960.
11. FAGAN, K. *What happens to your body when you've been in virtual reality for too long.* [online]. 2018 [visited on 2020-05-04]. Available from: https://www.businessinsider.com/virtual-reality-vr-side-effects-2018-3.
12. *HTC Vive Controllers: HTC Vive controller description* [online]. 2017 [visited on 2020-04-29]. Available from: https://www.en.magicgameworld.com/wp-content/uploads/2017/10/IMG_7953.jpg.
13. *BlenderXR* [online]. MARUI-PlugIn (inc.), 2018 [visited on 2020-05-04]. Available from: https://github.com/MARUI-PlugIn/BlenderXR/.

14. *SteamVR* [online]. Valve Corporation, 2020 [visited on 2020-05-04]. Available from: https://www.steamvr.com/.

15. *Visual Studio Code*. Microsoft Corporation, 2020. Available also from: https://code.visualstudio.com/.

16. SELLERS, G.; WRIGHT, R.S.; HAEMEL, N. *OpenGL SuperBible: Comprehensive Tutorial and Reference*. Pearson Education, 2013. OpenGL. ISBN 9780133365085. Available also from: https://books.google.cz/books?id=odgdAAAAQBAJ.

17. *Blender OpenGL Wrapper*. Blender Foundation, 2020. Available also from: https://docs.blender.org/api/current/bgl.html.

18. PALMÉR, Moses. *Pynput library for controlling and monitoring input devices*. 2020. Available also from: https://pynput.readthedocs.io/en/latest/.

A Add-on setup

For my add-on to work correctly, there are some steps and setup necessary to be done. This is the setup that should make my add-on work properly in Windows 10 (Version 1903). The versions of the selected libraries were tested to be working.

1. Download BlenderXR. (`https://www.marui-plugin.com/blender-xr/#download` or `https://github.com/MARUI-PlugIn/BlenderXR`, version available on 2020-04-06.)
2. Download files provided with the thesis.
3. Add `BlenderXR_SteamVR.dll` and `openvr_api.dll` into the root folder of BlenderXR.
4. Download pyopenvr from (`https://github.com/cmbruns/pyopenvr`, version available on 2020-03-29) and put the folder named openvr found in `pyopenvr-master\pyopenvr-master\src` in the `PathToBlenderXR\blenderversion\scripts\modules`.
5. Download pynput from (`https://pypi.org/project/pynput/#files`, version 1.6.8) and put the folder named pynput found in `pynput-1.6.8\pynput-1.6.8\lib` in the `PathToBlenderXR\blenderversion\scripts\modules`.
6. download `six` library from (`https://pypi.org/project/six/#files`, version 1.14.0) and put the `six.py` file found in directory `six-1.14.0\six-1.14.0` in the `PathToBlenderXR\blenderversion\scripts\modules`.
7. Move the folder named models from the files provided with the thesis to the root folder of BlenderXR.
8. Start BlenderXR.
9. Install the add-on (Edit -> Preferences -> Add-ons -> install (in the top-right corner) -> find the zip folder named Addon in provided files and select it -> install add-on).
10. Enable the add-on by clicking on the checkbox of the add-on.
11. Check if you have already set up (Room setup, plugged in base stations etc.) the VR system (HTC Vive in my case).
12. Start the add-on from the right sidebar in the 3D view.
13. Start a new BlenderXR window (Window -> New VR Window).
14. Put the headset on your head and grab the controllers.